A Simple Imperative Language
Operational Semantics
(= “meaning”)

Survey Results: Goals
• How PL relates to security (2)
• Type systems and theory (2)
• Get the basics of PL (2)
• New languages
• Symbolic execution
• Abstract interpretation
• Theorem proving
• Find a research topic
• Understand the CQual paper
• Help with Quals
• Advanced Topics

Homework #1 Out Today
• Due Tuesday, Jan 31 (1 week from now)
• Take a look tonight
• My office hours are on Wednesday

Today’s Plan
• Study a simple imperative language IMP
  - Abstract syntax
  - Operational semantics
  - Denotational semantics
  - Axiomatic semantics
  - ... and relationships between various semantics (with proofs, peut-être)
  - Today: operational semantics
    • (Chapter 2 of Winskel)

Some Survey Results

| 12. 2. 1. | I have taken a course that covered induction. I am comfortable proving things using induction. |
| 6. 3. 6. | I am comfortable with a functional programming language (e.g., LISP, Scheme, ML, or even Python). |
| 6. 0. 9. | I have used an “automated” bug-finding tool (e.g., FindBugs, PREfast, ESC/Java, JLint, PMD, Fortify, LCLint, Coverity, etc.). |
| 4. 4. 7. | I can typeset documents in LaTeX. |
| 2. 1. 12. | I have written a compiler that had a type checker. |

Syntax of IMP
• Concrete syntax
  - The rules by which programs can be expressed as strings of characters
  - Keywords, identifiers, statement separators (terminators), comments, indentation, etc.
• Concrete syntax is important in practice
  - For readability, familiarity, parsing speed, effectiveness of error recovery, clarity of error messages
• Well understood principles
  - Use finite automata and context-free grammars
  - Automatic lexer/parser generators
(Note On Recent Research)

• If-as-and-when you find yourself making a new language, consider GLR (elkhound) instead of LALR(1) (bison)
• Scott McPeak, George G. Necula: Elkhound: A Fast, Practical GLR Parser Generator. CC 2004: pp. 73-88
• As fast as LALR(1), more natural, handles basically all of C++, etc.

Abstract Syntax

• We ignore parsing issues and study programs given as abstract syntax trees
• Abstract syntax tree is (a subset of) the parse tree of the program
  - Ignores issues like comment conventions
  - More convenient for formal and algorithmic manipulation

IMP Abstract Syntactic Entities

• int integer constants \((n \in \mathbb{Z})\)
• bool boolean constants (true, false)
• L locations of variables \((x, y)\)
• Aexp arithmetic expressions \((e)\)
• Bexp boolean expressions \((b)\)
• Com commands \((c)\)
  - (these also encode the types)

Abstract Syntax (Aexp)

• Arithmetic expressions (Aexp)
  \[
  e ::= \begin{array}{ll}
  n & \text{for } n \in \mathbb{Z} \\
  x & \text{for } x \in L \\
  e_1 + e_2 & \text{for } e_1, e_2 \in Aexp \\
  e_1 - e_2 & \text{for } e_1, e_2 \in Aexp \\
  e_1 \times e_2 & \text{for } e_1, e_2 \in Aexp
  \end{array}
  \]

• Notes:
  - Variables are not declared
  - All variables have integer type
  - No side-effects (in expressions)

Abstract Syntax (Bexp)

• Boolean expressions (Bexp)
  \[
  b ::= \begin{array}{ll}
  \text{false} & \\
  e_1 = e_2 & \text{for } e_1, e_2 \in Aexp \\
  e_1 \leq e_2 & \text{for } e_1, e_2 \in Aexp \\
  \neg b & \text{for } b \in Bexp \\
  b_1 \land b_2 & \text{for } b_1, b_2 \in Bexp \\
  b_1 \lor b_2 & \text{for } b_1, b_2 \in Bexp
  \end{array}
  \]

“Boolean”

• George Boole
  - 1815-1864
• I’ll assume you know boolean algebra …
Abstract Syntax (Com)

- Commands (Com)
  \[
  c ::= \text{skip} \quad | \quad x := e \quad | \quad c_1 ; c_2 \quad | \quad \text{if } b \text{ then } c_1 \text{ else } c_2 \quad | \quad \text{while } b \text{ do } c
  \]
  where:
  - \( x \in \text{L} \land e \in \text{Aexp} \)
  - \( c_1, c_2 \in \text{Com} \land b \in \text{Bexp} \)

- Notes:
  - The typing rules have been embedded in the syntax definition
  - Other parts are not context-free and need to be checked separately (e.g., all variables are declared)
  - Commands contain all the side-effects in the language
  - Missing: pointers, function calls, what else?

Why Study Formal Semantics?

- Language design (denotational)
- Proofs of correctness (axiomatic)
- Language implementation (operational)
- Reasoning about programs
- Providing a clear behavioral specification
  - “All the cool people are doing it.”
    - You need this to understand PL research
  - “First one’s free.”

Consider This Java

\[
x = 0;
\text{try} \{
  x = 1;
  \text{break } \text{mygoto};
\}\text{finally} \{
  x = 2;
  \text{raise } \text{NullPointerException};
\}
x = 3;
\text{mygoto}:
x = 4;
\]

- What happens when you execute this code?
- Notably, what assignments are executed?

Why Study Formal Semantics?

- Language design (denotational)
- Proofs of correctness (axiomatic)
- Language implementation (operational)
- Reasoning about programs
- Providing a clear behavioral specification
  - “All the cool people are doing it.”
    - You need this to understand PL research
  - “First one’s free.”

Consider This Java

\[
x = 0;
\text{try} \{
  x = 1;
  \text{break } \text{mygoto};
\}\text{finally} \{
  x = 2;
  \text{raise } \text{NullPointerException};
\}
x = 3;
\text{mygoto}:
x = 4;
\]

- What happens when you execute this code?
- Notably, what assignments are executed?

14.20.2 Execution of try-catch-finally

- A try statement with a finally block is executed by first executing the try block. Then there is a choice:
  - If execution of the try block completes normally, then the finally block is executed, and then:
    - If the finally block completes normally, then the try statement completes normally.
  - If the try block completes abnormally because of a throw of a value \( V \), then there is a choice:
    - If the run-time type of \( V \) is assignable to the parameter of any catch clause of the try statement, then:
      - The first (leftmost) such catch clause is selected. The value \( V \) is assigned to the parameter of the selected catch clause, and the block of that catch clause is executed. Then there is a choice:
        - If the catch block completes normally, then the finally block is executed. Then there is a choice:
          - If the finally block completes normally, then the try statement completes normally.
          - If the finally block completes abnormally for reason \( R \), then the try statement completes abnormally for reason \( R \).
        - If the catch block completes abnormally because of a throw of value \( V \), then:
          - The first (leftmost) such catch clause is selected. The value \( V \) is assigned to the parameter of the selected catch clause, and the block of that catch clause is executed. Then there is a choice:
            - If the catch block completes normally, then the finally block is executed. Then there is a choice:
              - If the finally block completes normally, then the try statement completes normally.
              - If the finally block completes abnormally for reason \( S \), then the try statement completes abnormally for reason \( S \).
    - If the run-time type of \( V \) is not assignable to the parameter of any catch clause of the try statement, then:
      - The finally block is executed. Then there is a choice:
        - If the finally block completes normally, then the try statement completes abnormally for reason \( R \).
        - If the finally block completes abnormally because of a throw of value \( V \), then:
          - If the run-time type of \( V \) is not assignable to the parameter of any catch clause of the try statement, then:
            - The finally block is executed. Then there is a choice:
              - If the finally block completes normally, then the try statement completes abnormally for reason \( R \).
              - If the finally block completes abnormally for reason \( S \), then the try statement completes abnormally for reason \( S \).

Ouch!

- Wouldn’t it be nice if we had some way of describing what a language (feature or program) means ...  
  - More precisely than English
  - More compactly than English
  - So that you might build a compiler
  - So that you might prove things about programs

Popular Culture

“Ah. You seek meaning.”
Yes.
“Then listen to the music, not the song.”
― Kosh and Talia, Deathwalker

“Angel... How did you get in here?”
‘I was invited. The sign in front of the school... Formeia trans sicere educatum.’
“Enter all ye who seek knowledge.”
‘What can I say? I’m a knowledge seeker.’
― Jenny Calendar and Angelus, Passion
Analysis of IMP

• Questions to answer:
  – What is the “meaning” of a given IMP expression/command?
  – How would we go about evaluating IMP expressions and commands?
  – How are the evaluator and the meaning related?

Three Canonical Approaches

• Operational
  – How would I execute this?
  – “Symbolic Execution”

• Axiomatic
  – What is true after I execute this?

• Denotational
  – What is this trying to compute?

An Operational Semantics

• Specifies how expressions and commands should be evaluated

• Operational semantics abstracts the execution of a concrete interpreter

• Depending on the form of the expression
  – 0, 1, 2, . . . don’t evaluate any further.
  – They are normal forms or values.
  – \( e_1 + e_2 \) is evaluated by first evaluating \( e_1 \) to \( n_1 \), then evaluating \( e_2 \) to \( n_2 \) (post-order traversal)
  – The result of the evaluation is the literal representing \( n_1 + n_2 \).
  – Similarly for \( e_1 \times e_2 \)

Semantics of IMP

• The meaning of IMP expressions depends on the values of variables
  – What does “x+5” mean? It depends on “x”!

• The value of variables at a given moment is abstracted as a function from \( L \) to \( \mathbb{Z} \) (a state)
  – If \( x \mapsto 8 \) in our state, we expect “x+5” to mean 13

• The set of all states is \( \Sigma = L \rightarrow \mathbb{Z} \)

• We shall use \( \sigma \) to range over \( \Sigma \)
  – \( \sigma \), a state, maps variables to values

Notation: Judgment

• We write:

\[ <e, \sigma> \Downarrow n \]

• To mean that \( e \) evaluates to \( n \) in state \( \sigma \).

• This is a judgment. It asserts a relation between \( e \), \( \sigma \) and \( n \).

• In this case we can view \( \Downarrow \) as a function with two arguments (\( e \) and \( \sigma \)).

Operational Semantics

• This formulation is called natural operational semantics
  – or big-step operational semantics
  – the judgment relates the expression and its “meaning”

• How should we define

\[ <e_1 + e_2, \sigma> \Downarrow \ldots ? \]
Notation: Rules of Inference

• We express the evaluation rules as rules of inference for our judgment
  - called the derivation rules for the judgment
  - also called the evaluation rules (for operational semantics)
• In general, we have one rule for each language construct:
  \[
  \frac{<e_1, \sigma> \Downarrow n_1 \quad <e_2, \sigma> \Downarrow n_2}{<e_1 + e_2, \sigma> \Downarrow n_1 + n_2}
  \]

Rules of Inference

Hypothesis_1 ... Hypothesis_n

Conclusion

\[
\Gamma \vdash b : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau
\]

\[
\Gamma \vdash \text{if } b \text{ then } e_1 \text{ else } e_2 : \tau
\]

• For any given proof system, a finite number of rules of inference (or schema) are listed somewhere
• Rule instances should be easily checked
• What is the definition of “NP”?  

Derivation

- Tree-structured (conclusion at bottom)
- May include multiple sorts of rules of inference
- Could be constructed, typically are not
- Typically verified in polynomial time

Evaluation Rules (for Aexp)

<table>
<thead>
<tr>
<th>[&lt;n, \sigma&gt; \Downarrow n]</th>
<th>[&lt;x, \sigma&gt; \Downarrow \sigma(x)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&lt;e_1, \sigma&gt; \Downarrow n_1 \quad &lt;e_2, \sigma&gt; \Downarrow n_2]</td>
<td>[&lt;e_1, \sigma&gt; \Downarrow n_1 \quad &lt;e_2, \sigma&gt; \Downarrow n_2]</td>
</tr>
<tr>
<td>[&lt;e_1 + e_2, \sigma&gt; \Downarrow n_1 + n_2]</td>
<td>[&lt;e_1 - e_2, \sigma&gt; \Downarrow n_1 - n_2]</td>
</tr>
<tr>
<td>[&lt;e_1 \ast e_2, \sigma&gt; \Downarrow n_1 \ast n_2]</td>
<td>[&lt;e_1 \ast e_2, \sigma&gt; \Downarrow n_1 \ast n_2]</td>
</tr>
</tbody>
</table>

• This is called structural operational semantics
  - rules defined based on the structure of the expression
  - These rules do not impose an order of evaluation!

Evaluation Rules (for Bexp)

<table>
<thead>
<tr>
<th>[&lt;\text{true}, \sigma&gt; \Downarrow \text{true}]</th>
<th>[&lt;\text{false}, \sigma&gt; \Downarrow \text{false}]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&lt;b_1, \sigma&gt; \Downarrow \text{false}]</td>
<td>[&lt;b_2, \sigma&gt; \Downarrow \text{false}]</td>
</tr>
<tr>
<td>[&lt;b_1 \land b_2, \sigma&gt; \Downarrow \text{false}]</td>
<td>[&lt;b_1 \land b_2, \sigma&gt; \Downarrow \text{false}]</td>
</tr>
<tr>
<td>[&lt;b_1, \sigma&gt; \Downarrow \text{true}]</td>
<td>[&lt;b_2, \sigma&gt; \Downarrow \text{true}]</td>
</tr>
<tr>
<td>[&lt;b_1 \land b_2, \sigma&gt; \Downarrow \text{true}]</td>
<td>[&lt;b_1 \land b_2, \sigma&gt; \Downarrow \text{true}]</td>
</tr>
</tbody>
</table>

(show: possible \lor rule)

How to Read the Rules?

• Forward (top-down) = inference rules
  - if we know that the hypothesis judgments hold then we can infer that the conclusion judgment also holds

- If we know that \[<e_1, \sigma> \Downarrow 5\] and \[<e_2, \sigma> \Downarrow 7\], then we can infer that \[<e_1 + e_2, \sigma> \Downarrow 12\]
How to Read the Rules?

- **Backward (bottom-up) = evaluation rules**
  - Suppose we want to evaluate $e_1 + e_2$, i.e., find $n$ s.t. $e_1 + e_2 \Downarrow n$ is derivable using the previous rules
  - By inspection of the rules we notice that the last step in the derivation of $e_1 + e_2 \Downarrow n$ must be the addition rule
  - the other rules have conclusions that would not match $e_1 + e_2 \Downarrow n$
  - this is called reasoning by **inversion** on the derivation rules

Evaluation By Inversion

- Thus we must find $n_1$ and $n_2$ such that $e_1 \Downarrow n_1$ and $e_2 \Downarrow n_2$ are derivable
  - This is done recursively
- If there is exactly one rule for each kind of expression we say that the rules are **syntax-directed**
  - At each step at most one rule applies
  - This allows a simple evaluation procedure as above
  - True for our $A_{exp}$ but not $B_{exp}$. Why?

Evaluation of Commands

- The evaluation of a Com may have side effects but has **no direct result**
  - What is the result of evaluating a command?
- The "result" of a Com is a new state:
  
  $<c, \sigma> \Downarrow \sigma'$

  - But the evaluation of Com might not terminate! Danger Will Robinson!

Com Evaluation Rules 1

- $<\text{skip}, \sigma> \Downarrow \sigma$
- $<\text{skip}, \sigma> \Downarrow \sigma$
- $<\text{if } b \text{ then } c_1 \text{ else } c_2, \sigma> \Downarrow \sigma'$
- $<\text{if } b \text{ then } c_1 \text{ else } c_2, \sigma> \Downarrow \sigma'$

Com Evaluation Rules 2

- $<x := e, \sigma> \Downarrow \sigma[x := n]$

  - Let's do **while** together

Com Evaluation Rules 3

- $<x := e, \sigma> \Downarrow \sigma[x := n]$

  - Def: $\sigma[x := n](x) = n$
  - $\sigma[x := n](y) = \sigma(y)$

  - $<b, \sigma> \Downarrow \text{false}$
  - $<\text{while } b \text{ do } c, \sigma> \Downarrow \sigma'$
  - $<b, \sigma> \Downarrow \text{true}$
  - $<c; \text{ while } b \text{ do } c, \sigma> \Downarrow \sigma'$
Homework

- Homework 1 Out Today
  - Actually out last Friday ...
  - Due Tuesday, January 31
- Read at least 1 of these 3 Articles
  - 1. Wegner’s Programming Languages - The First 25 years
  - 2. Wirth’s On the Design of Programming Languages
  - 3. Nauer’s Report on the algorithmic language ALGOL 60
- Skim the optional reading - we’ll discuss opsem “in the wild” next time